

Department I - C Plus Plus

Modern and Lucid C++  
for Professional Programmers

Week 6 – Namespaces and Enums

Thomas Corbat / Felix Morgner  
Rapperswil, 20.10.2020  
HS2020



INSTITUTE FOR  
SOFTWARE

- You can to group and structure names into namespaces
- You can resolve function calls using Argument Dependent Lookup
- You can create enumerations as simple types with few values
- You know how to implement your own arithmetic type and are aware of possible amibiguities

- **Recap Week 5**
- **Namespaces**
- **Enumerations**
- **Arithmetic Types**

## Recap Week 5



- What is wrong with the following constructor implementation?

```
Date::Date(int year, int month, int day) {  
    this->year = year;  
    this->month = month;  
    this->day = day;  
}
```

- How would it be correct?

```
Date::Date(int year, int month, int day)  
    : year{year}, month{month}, day{day} {  
}
```

- **How do you define an output operator for your own type**

- Always as free operator: Either with a helper member function ...

```
class Date {  
    int year, month, day;  
public:  
    std::ostream & print(std::ostream & os) const {  
        //print logic  
    }  
};  
inline std::ostream & operator<<(std::ostream & os, Date const & date) {  
    return date.print(os);  
}
```

- ... or as friend

```
class Date {  
    //...  
    friend std::ostream & operator<<(std::ostream & os, Date const & date);  
};
```

# Namespaces



## Goals:

- You can to group and structure names into namespaces
- You know how Argument Dependent Lookup works

- **Namespaces are scopes for grouping and preventing name clashes**
  - Same name in different scopes possible
  - Example `boost::optional` and `std::optional` can coexist
- **Global namespace has the `::` prefix**
  - Can be omitted if unique
  - `::std::cout` is usually equal to `std::cout`
- **Nesting of namespaces is possible**
  - Example: `std::literals::chrono_literals`
- **Nesting of scopes allows hiding of names**
  - Hidden names can only be accessed when they belong to a named scope



- Namespaces can only be defined outside of classes and functions
- The same namespace can be opened and closed multiple times to gather definitions and declarations (header files)
- Qualified names are used to access names in a namespace

■ `demo::subdemo::foo()`

- A name with a leading `::` is called a fully qualified name

■ `::std::cout`

```
namespace demo {
void foo(); //1
namespace subdemo {
void foo() { /*2*/ }
} // subdemo
} // demo

namespace demo {
void bar() {
    foo(); //1
    subdemo::foo(); //2
}
}

void demo::foo() { /*1*/ } // definition

int main() {
    using demo::subdemo::foo;
    foo(); //2
    demo::foo(); //1
    demo::bar();
}
```

- **Import a name from a namespace into the current scope**

- That name can be used without a namespace prefix
- Useful if the name is used very often

```
using std::string;  
string s{"no std::"};
```

- **Alternative: Using alias for types if a name is long**

```
using input = std::istream_iterator<int>;  
input eof{};
```

- **There are also using directives, which import ALL names of a namespace into the current scope**

- Use them only in local scope to avoid "pollution" of your namespace

```
int main() {  
    using namespace std;  
    cout << "Hello John";  
    //many more uses of std names  
}
```

- **Special case: omit name after namespace**

- The compiler chooses a unique internal identifier

- **Implicit using directive for the chosen name**

- **Hides module internals**

- Helper functions and types
- Constants

- **Use them only in source files (\*.cpp)**

```
#include <iostream>
namespace { // anonymous
void doit() { // can not be called
              // outside this file
    std::cout << "doit called\n";
}
} // anonymous namespace ends

void print() { // callable from other
              // parts if declared
    doit();
    std::cout << "print called\n";
}
```

```
void caller() { // in a different file
    void print(); // declare print
    print();
    void doit(); // declare doit
    doit();      // linker error ⚡
}
```

- **Class Date can/should be put in a namespace to group it with its operators and functions**

```
namespace calendar {  Date.h
class Date {
    int year, month, day;
public:
    Date tomorrow() const;
};
}
```

- **Using names from a namespace requires qualification**

```
calendar::Date d{};
```

- **Member implementations need to be qualified with that namespace too**

- Example: `calendar::Date::tomorrow`

```
namespace calendar {  
    class Date {  
        int year, month, day;  
    public:  
        Date tomorrow() const;  
    };  
}
```

Date.h

```
#include "Date.h"  
  
Date calendar::Date::tomorrow() const {  
    //...  
}
```

Date.cpp

- **Types and (non-member) functions belonging to that type should be placed in a common namespace**

```
namespace calendar { ... }
```

- **Advantage: Argument Dependent Lookup!**
  - When the compiler encounters an unqualified function or operator call with an argument of a user-defined type it looks into the namespace in which that type is defined to resolve the function/operator
  - E.g. it is not necessary to write `std::` in front of `for_each` when `std::vector::begin()` is an argument of the function

```
namespace calendar {                                     Date.h

class Date {
    //...
};

bool isHoliday(Date const &);

}
```

```
#include "Date.h"                                       Holidays.cpp

using Dates = std::vector<calendar::Date>;

void markHolidays(Dates const & dates) {
    for_each(begin(dates), end(dates),
        [](calendar::Date const & d) {
            if (isHoliday(d)) //calendar::isHoliday
                //...
        });
}
```

- Unqualified operator calls don't allow explicit namespace qualification

- NOT: `std::cout << calendar::<< birthday;` ⚡

- Functions and operators are looked up in the namespace of the type of their arguments first

- E.g. namespaces `std` and `calendar` for `std::cout << birthday;`

```
namespace calendar {  
class Date {  
    //...  
    bool operator<(Date const & rhs) const;  
};
```

```
inline bool operator>(Date const & lhs, Date const & rhs) {  
    return rhs < lhs;  
}  
}
```

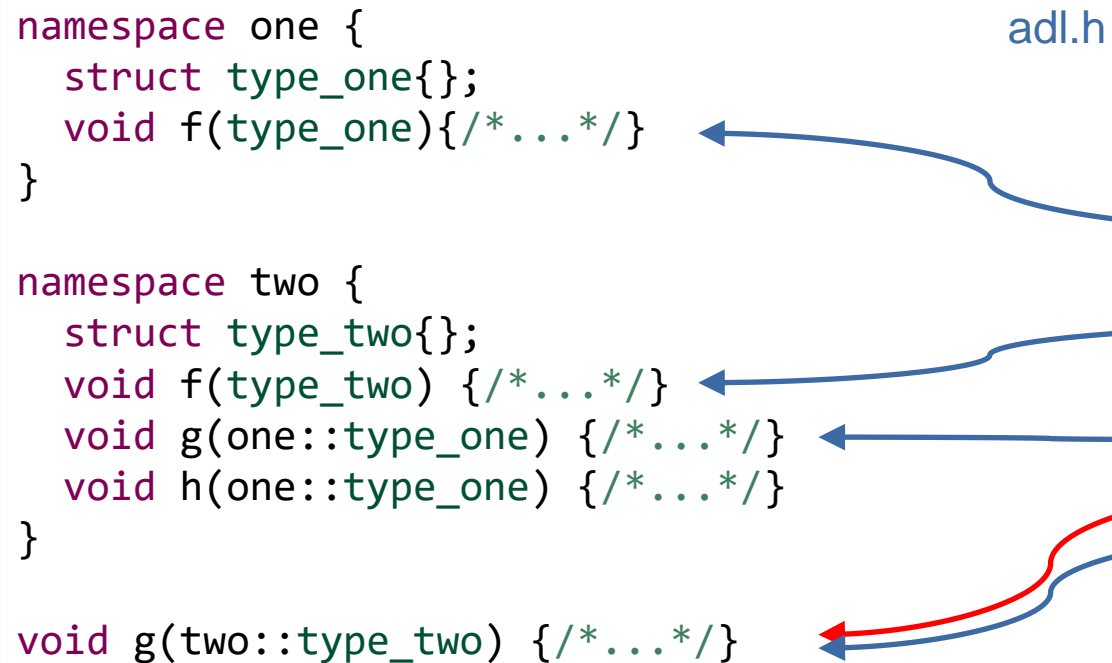
```
#include "Date.h"
```

```
bool millenial(calendar::Date birthday) {  
    calendar::Date lower{1981, 1, 1};  
    calendar::Date upper{1996, 12, 31};  
    return birthday > lower && birthday < upper;  
}
```

Date.h

adl.h

```
namespace one {  
    struct type_one{};  
    void f(type_one){/*...*/}  
}  
  
namespace two {  
    struct type_two{};  
    void f(type_two) {/*...*/}  
    void g(one::type_one) {/*...*/}  
    void h(one::type_one) {/*...*/}  
}  
  
void g(two::type_two) {/*...*/}
```



adl.cpp

```
#include "adl.h"  
  
int main() {  
    one::type_one t1{};  
    f(t1);  
    two::type_two t2{};  
    f(t2);  
    h(t1);  
    two::g(t1);  
    g(t1); //Argument type does not match  
    g(t2);  
}
```



- **Generic code (Templates) might not pick up a global operator<< in an algorithm call using ostream\_iterator if the value output is from namespace std too**

- E.g. `std::vector<int>`

- **This example only works when the operator<<(ostream &, vec const &) is put in namespace std, because both arguments are in std**

- **This is not allowed by the standard!**

```
using std::vector;
using std::ostream;
using vec = vector<int>;
using outv = std::ostream_iterator<vec>;
using out = std::ostream_iterator<int>;

namespace std {
ostream & operator<<(ostream & os, vec const & v) {
    copy(begin(v), end(v), out{os, ","});
    return os;
}
}

void worksOnlyWithOutputOpInNsStd(ostream & os) {
    vector<vec> vv{{1, 2, 3}, {4, 5, 6}};
    copy(begin(vv), end(vv), outv{os, "\n"});
}
```

intuition.cpp

- Create a new type by inheriting from `std::vector<int>`

- Requires support for inheriting constructors

```
struct Sub : Base {  
    using Base::Base;  
};
```

- Alias would not be sufficient

```
using vec = vector<int>;
```

- Not recommended to derive from standard containers in general
- Does not help with `std::map/std::pair`

```
using std::ostream;                                workaround.cpp  
using std::vector;  
using out = std::ostream_iterator<int>;  
  
namespace X {  
    struct vec : vector<int> { // vec is a new type  
        using vector<int>::vector; // inherit ctors  
    };  
    ostream & operator<<(ostream & os, vec const & v) {  
        copy(begin(v), end(v), out{os, ","});  
        return os;  
    }  
}  
  
void works_with_inheriting_ctors(ostream & os) {  
    using outv = std::ostream_iterator<X::vec>;  
    vector<X::vec> vv{{1, 2, 3},{4, 5, 6}};  
    copy(begin(vv), end(vv), outv{os, "\n"});  
}
```

```
namespace seasons {  
    struct Spring{/*...*/};  
}  
  
namespace waters {  
    struct Spring{/*...*/};  
}  
  
namespace machinery {  
    struct Spring{/*...*/};  
}
```

Correct

The same name, here Spring, can be defined multiple times if in different scopes. Namespaces open such scopes and further distinguish the names they contain. There are three qualified type names here:

```
::seasons::Spring  
::waters::Spring  
::machinery::Spring
```

```
namespace garden {  
    struct Plant{/*...*/};  
    void water(Plant &){/*...*/}  
}  
  
int main() {  
    garden::Plant geranium{};  
    water(geranium);  
}
```

Correct

The call `water(geranium)` does not need to be qualified by `garden::`, due to argument dependent lookup. Because the argument's type is defined in namespace `garden` and the call is unqualified, that namespace is considered for lookup as well.

# Enums



## Goals:

- You can create enumerations as simple types with few values
- You know the difference between scoped and unscoped enums
- You know how to specify the values of enumerators

- Enumerations are useful to represent types with only a few values

```
enum <name> {  
    <enumerators>  
};
```

```
enum class <name> {  
    <enumerators>  
};
```

- An enumeration creates a new type that can easily be converted to an integral type (unscoped enumeration only)
  - Conversion from an integral type to an enumeration is not possible implicitly
- The individual values (enumerators) are specified in the type
- Unless specified explicitly, the values start with 0 and increase by 1

- **Unscoped enumeration (no class keyword)**

```
enum DayOfWeek {  
    Mon, Tue, Wed, Thu, Fri, Sat, Sun  
}; 0    1    2    3    4    5    6
```

- Implicit conversion to int

```
int day = Sun;
```

- **Scoped enumeration (class keyword)**

```
enum class DayOfWeek {  
    Mon, Tue, Wed, Thu, Fri, Sat, Sun  
}; 0    1    2    3    4    5    6
```

- No implicit conversion to int, requires `static_cast`

```
int day = static_cast<int>(DayOfWeek::Sun);
```

- Conversion from int to enum always requires a `static_cast`

```
DayOfWeek tuesday = static_cast<DayOfWeek>(1);
```

- **Unscoped enumeration (no class keyword)**

- Enumerators leak into surrounding scope
- Best used as member of a class

```
namespace calendar {  
  
    enum DayOfWeek {  
        Mon, Tue, Wed, Thu, Fri, Sat, Sun  
    };  
    //Enumerators are visible here  
}  
  
bool is_weekend(calendar::DayOfWeek day) {  
    return day == calendar::Sat ||  
           day == calendar::Sun;  
}
```

- **Scoped enumeration (class keyword)**

- Enumerators do not leak into surrounding scope
- DayOfWeek:: qualification required

```
namespace calendar {  
  
    enum class DayOfWeek {  
        Mon, Tue, Wed, Thu, Fri, Sat, Sun  
    };  
}  
  
bool is_weekend(calendar::DayOfWeek day) {  
    return day == calendar::DayOfWeek::Sat ||  
           day == calendar::DayOfWeek::Sun;  
}
```

- Operators can be overloaded for **enums**

- Examples

- Prefix increment

```
DayOfWeek operator++(DayOfWeek &)
```

- Postfix increment

```
DayOfWeek operator++(DayOfWeek &, int)
```

```
enum DayOfWeek {
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
};

DayOfWeek operator++(DayOfWeek & aday) {
    int day = (aday + 1) % (Sun + 1);
    aday = static_cast<DayOfWeek>(day);
    return aday;
}

DayOfWeek operator++(DayOfWeek & aday, int) {
    DayOfWeek ret{aday};
    if (aday == Sun) {
        aday = Mon;
    } else {
        aday = static_cast<DayOfWeek>(aday + 1);
    }
    return ret;
}
```



- **With = values can be specified for enumerators**
  - Subsequent enumerators get value incremented (+1)
- **Different enumerators can have the same value**
- **In some cases enumerations are used to create bit masks**
  - Values are a power of 2 ( $2^x$ )

```
enum Month {  
    jan = 1, feb, mar, apr, may,  
    jun, jul, aug, sep, oct, nov, dec,  
    january = jan, february, march,  
    april, june = jun, july, august,  
    september, october, november,  
    december  
};
```

```
enum FilePermissions {  
    readable = 1,    //001  
    writeable = 2,   //010  
    executable = 4   //100  
};
```

- **Enumerator names are not mapped automatically to their original name**
  - Might become a feature in a future C++ standard (2023?)
- **You can provide a lookup table and overload the output operator (<<)**

```
std::ostream & operator<<(std::ostream & out, Month m) {  
    static std::array<std::string, 12> const monthNames {  
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",  
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };  
    out << monthNames[m - 1]; //m - 1 if Jan has value 1  
    return out;  
}
```

- Enumerations can specify the underlying type by inheritance
- The underlying type can be any integral type
- This allows forward-declaring enumerations
  - Can be used to hide implementation details if defined as a class member


```
enum class LaunchPolicy
: unsigned char
{
    sync = 1,
    async = 2,
    gpu = 4,
    process = 8,
    none = 0
};
```

```
#ifndef STATEMACHINE_H_
#define STATEMACHINE_H_

struct Statemachine {
    Statemachine();
    void processInput(char c);
    bool isDone() const;
private:
    enum class State : unsigned short;
    State theState;
};

#endif /* STATEMACHINE_H_ */
```

```
#include "Statemachine.h"
#include <cctype>
enum class Statemachine::State : unsigned short {
    begin, middle, end
};
Statemachine::Statemachine()
    : theState {State::begin} {}
void Statemachine::processInput(char c) {
    switch (theState) {
        case State::begin:
            if (!isspace(c)) {theState = State::middle;}
            break;
        case State::middle:
            if (isspace(c)) {theState = State::end;}
            break;
        case State::end:
            break; // ignore input
    }
}
bool Statemachine::isDone() const {
    return theState == State::end;
}
```



```
enum class Glass {  
    Empty = 0, HalfEmpty,  
    HalfFull = HalfEmpty, Full  
};  
void drink(Glass glass) {  
    if (glass == Empty) {  
        throw DieOfThirstException{"Empty"};  
    }  
}
```

Incorrect

Since Glass is an scoped enum it is not possible to access Empty without the Glass:: qualifier.

```
enum TddCycle : int {  
    RedBar, GreenBar, Refactor  
};  
void progress(TddCycle & cycle) {  
    if (cycle == Refactor) {  
        cycle = RedBar;  
    } else {  
        int current = cycle;  
        cycle = current + 1;  
    }  
}
```

Incorrect

An int cannot be converted to an enum implicitly. A static\_cast would be required to get from int to TddCycle

# Arithmetic Types (Ring5)



Goal:

- You know how to implement your own arithmetic type
- You know the ambiguities that arise from implicit conversion

- **Disclaimer: You usually do not want to implement your own arithmetic types!**
  - Future C++ standards will provide additional arithmetic types, like unbounded large integer and rational
- **Here are the basics if you ever need them anyway**
- **Example: Modulo arithmetic**
  - Rings / Finite fields
  - Used in Cryptography

- **Ring5: Arithmetic Modulo 5**

```
struct Ring5 {  
    explicit Ring5(unsigned x = 0u)  
        : val{x % 5}{}  
    unsigned value() const {  
        return val;  
    }  
private:  
    unsigned val;  
};
```

- **Invariant:**
  - Member variable is in range  $0..4$
- **Accessor to value**
- **Constructor explicit?**

- **Arithmetic types must be equality comparable**

- `operator ==`
- CUTE requires this operator in `ASSERT_EQUAL`

```
void testValueCtorWithLargeInput() {  
    Ring5 four{19};  
    ASSERT_EQUAL(Ring5{4}, four);  
}
```

- **Boost can be used to get `!=` easily**

- Use `boost::equality_comparable`

```
struct Ring5 :  
    boost::equality_comparable<Ring5> {  
    bool operator==(Ring5 const & r) const {  
        return val == r.val;  
    }  
    //...  
};
```



- It might be convenient to have the output operator (<<) to print a Ring5
  - As always: Output operator NOT as a class member!
- CUTE requires the output operator for nice failure messages

```
std::ostream & operator<<(std::ostream & out, Ring5 const & r) {  
    out << "Ring5{" << r.value() << '}';  
    return out;  
}
```

```
void testOutputOperator() {  
    std::ostringstream out{};  
    out << Ring5{4};  
    ASSERT_EQUAL("Ring5{4}", out.str());  
}
```

- **Result must be in the range 0..4**
  - E.g.  $(4 + 4) = 8 \% 5 = 3$
- **Implement both operators yourself**
  - `operator+`
  - `operator+=`
- **(Or) Use boost**
  - Implement `operator+=`
  - Derive from `boost::addable<Ring5>`

```
struct Ring5 :  
    boost::equality_comparable<Ring5>,  
    boost::addable<Ring5> {  
    Ring5 operator+=(Ring5 const & r) {  
        val = (val + r.val) % 5;  
        return *this;  
    }  
    //...  
};
```

```
void testAdditionWrap() {  
    Ring5 four{4};  
    Ring5 three = four + four;  
    ASSERT_EQUAL(Ring5{3}, three);  
}
```

- **Example for self implemented multiplication**

- Operator \*= as member function
- Operator \* as free (inline) function
  - Pass left-hand operand by value

- **Modulo only needed in \*= Operator**

- Avoids code duplication

```
struct Ring5 :  
    boost::equality_comparable<Ring5>,  
    boost::addable<Ring5> {  
    Ring5 operator*=(Ring5 const & r) {  
        val = (val * r.val) % 5;  
        return *this;  
    }  
    //...  
};
```

```
inline Ring5 operator*(Ring5 l,  
                       Ring5 const & r) {  
    l *= r;  
    return l;  
}
```

- **What if we want to add Ring5 and int?**
- **Either implement all combinations of parameters for operator+**
  - `operator+(Ring5, unsigned)`
  - `operator+(unsigned, Ring5)`
- **Or make constructor non-explicit**
- **The latter might become a problem when we want to have automatic conversion to unsigned as well**
  - Ambiguity or wrong conversion happens

- four should have the value 4
- four should have the type Ring5

```
void test_AdditionWithInt_ValueIsFour() {
    Ring5 two{2};
    auto four = two + 2u;
    ASSERT_EQUAL(Ring5{4}, four);
}

void test_AdditionWithInt_TypeIsRing5() {
    Ring5 two{2};
    auto four = two + 2u;
    ASSERT_EQUAL(typeid(Ring5).name(),
        typeiddecltype(four).name());
}
```

- **Overhead with code, duplication, danger of wrong implementation**

- Test cases help
- Overload could be forgotten

```
inline Ring5 operator+(Ring5 const & l, unsigned r) {  
    return Ring5{l.value() + r};  
}  
  
inline Ring5 operator+(unsigned l, Ring5 const & r) {  
    return Ring5{l + r.value()};  
}
```

- **Non-explicit constructor provides automatic inward conversion**

- Type conversion operator
  - `operator <type>() const` member function
  - `explicit` preferred but requires `static_cast`

- **Danger: Ambiguities and unexpected conversions are lurking**

```
struct Ring5 {  
    Ring5(unsigned x) : val{ x % 5 } {}  
    operator unsigned() const {  
        return val;  
    }  
    //...  
};
```

- **Namespaces are used to group types and functions belonging together with a sensible name**
- **Argument dependent lookup allows unqualified function calls to find functions in the scope of its arguments type's scope**
- **Enums are used to create types with a small range of named values**
- **Arithmetic types have to be implemented with care due to automatic conversion**

- C++ allows to mark functions and constructors as constexpr
- Guarantees compiler to evaluate function
- Provides support for literals type that can be initialized at compile-time
- Functions must be "pure" functions
  - no external side-effects
  - no memory
  - no exceptions
- Faster program execution

```
struct Ring5 {  
    explicit constexpr Ring5(unsigned x)  
        : val{ x % 5 } {  
    }  
    constexpr Ring5 operator+(Ring5 const & r) const {  
        return Ring5{val + r.val};  
    }  
    //...  
};
```

- Writing `Ring5{4}` to create a value of type `Ring5` is a bit annoying when many such values are needed
- `4_r5` would be shorter
- User-defined literals allow to defined suffixes to constants to change their type or value

```
constexpr <type> operator"" <suffix>(<parameter>);
```

- Must be put in a (separate) namespace

```
namespace R5 {  
    constexpr Ring5 operator"" _r5(unsigned long long v) {  
        return Ring5{static_cast<unsigned>(v % 5)};  
    }  
}
```

- Requires using namespace directive

```
using namespace R5;  
static_assert(Ring5{2} == 7_r5, "UDL operator");
```